RoboTwin 2.0

🤖 RoboTwin 机器人控制基础操作包

1. 核心控制函数

1.1 抓取操作（`grasp_actor`）

```
def grasp_actor(
    self,
    actor: Actor,                # 要抓取的物体
    arm_tag: ArmTag,               # 手臂标签 ("left" 或 "right")
    pre_grasp_dis=0.1,           # 预抓取距离
    grasp_dis=0,                 # 抓取距离
    gripper_pos=0.0,             # 夹爪位置
    contact_point_id: list | float = None,  # 接触点ID
):
    # 返回: (arm_tag, [Action列表])
```

1.2 放置操作（`place_actor`）

```
def place_actor(
    self,
    actor: Actor,                # 要放置的物体
    arm_tag: ArmTag,               # 手臂标签
    target_pose: list | np.ndarray, # 目标位置
    functional_point_id: int = None, # 功能点ID
    pre_dis: float = 0.1,          # 预放置距离
    dis: float = 0.02,           # 放置距离
    is_open: bool = True,          # 是否张开夹爪
    constrain: Literal["free", "align", "auto"] = "auto",  # 约束类型
    **args,
```

```
):
    # 返回: (arm_tag, [Action列表])
```

1.3 位移移动（`move_by_displacement`）

```
def move_by_displacement(
    self,
    arm_tag: ArmTag,              # 手臂标签
    x: float = 0.0,               # X轴位移
    y: float = 0.0,               # Y轴位移
    z: float = 0.0,               # Z轴位移
    quat: list = None,            # 四元数旋转
    move_axis: Literal["world", "arm"] = "world",  # 移动坐标系
):
    # 返回: (arm_tag, [Action])
```

1.4 姿态移动（`move_to_pose`）

```
def move_to_pose(
    self,
    arm_tag: ArmTag,              # 手臂标签
    target_pose: list | np.ndarray | sapien.Pose,  # 目标姿态
):
    # 返回: (arm_tag, [Action])
```

1.5 夹爪控制

```
def close_gripper(self, arm_tag: ArmTag, pos: float = 0.0):
    # 关闭夹爪，pos: 0.0=完全关闭, 1.0=完全张开
    return arm_tag, [Action(arm_tag, "close", target_gripper_pos=pos)]

def open_gripper(self, arm_tag: ArmTag, pos: float = 1.0):
```

```
    # 张开夹爪
    return arm_tag, [Action(arm_tag, "open", target_gripper_pos=pos)]
```

1.6 回到原点（`back_to_origin`）

```
def back_to_origin(self, arm_tag: ArmTag):
    # 返回手臂到初始位置
    return arm_tag, [Action(arm_tag, "move", self.robot.left/right_original_pose)]
```

2. 动作执行系统

2.1 动作类（`Action`）

```
class Action:
    def __init__(
        self,
        arm_tag: ArmTag | Literal["left", "right"],
        action: Literal["move", "open", "close", "gripper"],
        target_pose: sapien.Pose | list | np.ndarray = None,
        target_gripper_pos: float = None,
        **args,
    ):
```

2.2 手臂标签（`ArmTag`）

```
class ArmTag:
    def __init__(self, value):  # "left" 或 "right"
    @property
    def opposite(self):  # 获取相反的手臂
```

2.3 动作执行（`move`）

```
def move(
    self,
```

```
    actions_by_arm1: tuple[ArmTag, list[Action]],  # 第一个手臂的动作
    actions_by_arm2: tuple[ArmTag, list[Action]] = None,  # 第二个手臂的动作
    save_freq=-1,  # 保存频率
):
    # 执行动作序列，支持双臂协调
```

3. 高级控制功能

3.1 双臂协调移动

```
def together_move_to_pose(
    self,
    left_target_pose,              # 左臂目标位置
    right_target_pose,             # 右臂目标位置
    left_constraint_pose=None,     # 左臂约束
    right_constraint_pose=None,    # 右臂约束
    use_point_cloud=False,         # 是否使用点云
    use_attach=False,              # 是否使用附着
    save_freq=-1,                  # 保存频率
):
```

3.2 密集动作控制

```
def take_dense_action(self, control_seq, save_freq=-1):
    """
    control_seq: {
        "left_arm": {"position": [...], "velocity": [...]},
        "right_arm": {"position": [...], "velocity": [...]},
        "left_gripper": {"num_step": int, "target_pos": float},
        "right_gripper": {"num_step": int, "target_pos": float}
    }
    """
```

4. 状态查询函数

4.1 获取手臂姿态

```
def get_arm_pose(self, arm_tag: ArmTag):
    # 返回当前手臂末端执行器姿态
    if arm_tag == "left":
        return self.robot.get_left_ee_pose()
    elif arm_tag == "right":
        return self.robot.get_right_ee_pose()
```

4.2 夹爪状态检查

```
def is_left_gripper_open(self):
def is_right_gripper_open(self):
def is_left_gripper_close(self):
def is_right_gripper_close(self):
```

5. 规划器系统

5.1 运动规划器

- **MplibPlanner**: 基于mplib的规划器
- **CuroboPlanner**: 基于Curobo的规划器
- **TOPP**: 时间最优路径规划

5.2 规划参数

```
planner_type = "mplib_RRT"  # 规划器类型
plan_step_lim = 2500        # 规划步数限制
```

接口函数详细分析

1. 抓取操作 (`grasp_actor`)

输入参数详解

```
def grasp_actor(
    self,
    actor: Actor,                  # 要抓取的物体对象
    arm_tag: ArmTag,                 # 手臂选择 ("left" 或 "right")
    pre_grasp_dis=0.1,             # 预抓取距离 (米)
    grasp_dis=0,                   # 最终抓取距离 (米)
    gripper_pos=0.0,                 # 夹爪位置 (0.0=关闭, 1.0=张开)
    contact_point_id: list | float = None,  # 接触点索引
):
```

内部执行流程

```
def grasp_actor(self, actor, arm_tag, pre_grasp_dis=0.1, grasp_dis=0, gripper_
pos=0.0, contact_point_id=None):
    # 1. 检查规划是否成功
    if not self.plan_success:
        return None, []

    # 2. 计算抓取姿态
    pre_grasp_pose, grasp_pose = self.choose_grasp_pose(
        actor, arm_tag=arm_tag, pre_dis=pre_grasp_dis,
        target_dis=grasp_dis, contact_point_id=contact_point_id
    )

    # 3. 生成动作序列
    if pre_grasp_pose == grasp_pose:
        # 如果预抓取和抓取位置相同，直接抓取
        return arm_tag, [
            Action(arm_tag, "move", target_pose=pre_grasp_pose),
            Action(arm_tag, "close", target_gripper_pos=gripper_pos),
        ]
    else:
        # 否则分两步：先移动到预抓取位置，再移动到抓取位置
```

```
        return arm_tag, [
            Action(arm_tag, "move", target_pose=pre_grasp_pose),
            Action(arm_tag, "move", target_pose=grasp_pose, constraint_pose=[1,
1,1,0,0,0]),
            Action(arm_tag, "close", target_gripper_pos=gripper_pos),
        ]
```

抓取姿态计算（`choose_grasp_pose`）

```
def choose_grasp_pose(self, actor, arm_tag, pre_dis=0.1, target_dis=0, contact
_point_id=None):
    # 1. 获取物体当前姿态
    actor_pose = actor.get_pose()

    # 2. 根据接触点计算抓取位置
    if contact_point_id is not None:
        if isinstance(contact_point_id, (list, tuple)):
            # 多点接触：计算平均位置
            contact_points = [actor.get_contact_point(i) for i in contact_point_id]
            grasp_point = np.mean(contact_points, axis=0)
        else:
            # 单点接触
            grasp_point = actor.get_contact_point(contact_point_id)
    else:
        # 默认使用物体中心
        grasp_point = np.array([0, 0, 0])

    # 3. 转换到世界坐标系
    world_grasp_point = actor_pose.transform(grasp_point)

    # 4. 计算预抓取位置（沿Z轴向上偏移）
    pre_grasp_point = world_grasp_point + np.array([0, 0, pre_dis])

    # 5. 计算最终抓取位置
    final_grasp_point = world_grasp_point + np.array([0, 0, target_dis])
```

```
    # 6. 生成姿态（位置 + 四元数）
    pre_grasp_pose = [pre_grasp_point[0], pre_grasp_point[1], pre_grasp_point
[2], 1, 0, 0, 0]
    grasp_pose = [final_grasp_point[0], final_grasp_point[1], final_grasp_point
[2], 1, 0, 0, 0]

    return pre_grasp_pose, grasp_pose
```

2. 放置操作（`place_actor`）

输入参数详解

```
def place_actor(
    self,
    actor: Actor,                 # 要放置的物体
    arm_tag: ArmTag,              # 手臂标签
    target_pose: list | np.ndarray, # 目标位置 [x, y, z, qx, qy, qz, qw]
    functional_point_id: int = None, # 功能点ID（物体上的特定点）
    pre_dis: float = 0.1,         # 预放置距离
    dis: float = 0.02,            # 最终放置距离
    is_open: bool = True,         # 是否张开夹爪
    constrain: Literal["free", "align", "auto"] = "auto",  # 约束类型
    **args,
):
```

内部执行流程

```
def place_actor(self, actor, arm_tag, target_pose, functional_point_id=None, pr
e_dis=0.1, dis=0.02, is_open=True, **args):
    # 1. 计算放置姿态
    place_poses = self.get_place_pose(
        actor, arm_tag, target_pose,
        functional_point_id=functional_point_id,
        pre_dis=pre_dis, **args
```

```
    )

    # 2. 生成动作序列
    actions = []
    for pose in place_poses[:-1]:  # 除了最后一个位置
        actions.append(Action(arm_tag, "move", target_pose=pose))

    # 3. 最终放置动作
    final_pose = place_poses[-1]
    actions.append(Action(arm_tag, "move", target_pose=final_pose, constraint
_pose=[1,1,1,0,0,0]))

    # 4. 张开夹爪（如果需要）
    if is_open:
        actions.append(Action(arm_tag, "open", target_gripper_pos=1.0))

    return arm_tag, actions
```

放置姿态计算（`get_place_pose`）

```
def get_place_pose(self, actor, arm_tag, target_pose, constrain="auto", functi
onal_point_id=None, pre_dis=0.1, pre_dis_axis="grasp"):
    # 1. 获取目标位置
    if isinstance(target_pose, (list, np.ndarray)):
        target_pos = np.array(target_pose[:3])
        target_quat = np.array(target_pose[3:])
    else:
        target_pos = target_pose.p
        target_quat = target_pose.q

    # 2. 计算功能点偏移
    if functional_point_id is not None:
        fp_offset = actor.get_functional_point(functional_point_id)
        target_pos += fp_offset
```

```
    # 3. 计算预放置位置
    if pre_dis_axis == "fp":
        # 沿功能点方向偏移
        pre_pos = target_pos + np.array([0, 0, pre_dis])
    else:
        # 沿抓取方向偏移
        current_pose = self.get_arm_pose(arm_tag)
        grasp_direction = current_pose[:3] - target_pos
        grasp_direction = grasp_direction / np.linalg.norm(grasp_direction)
        pre_pos = target_pos + grasp_direction * pre_dis

    # 4. 生成姿态序列
    poses = []
    poses.append([pre_pos[0], pre_pos[1], pre_pos[2]] + target_quat.tolist())
    poses.append([target_pos[0], target_pos[1], target_pos[2]] + target_quat.tol
ist())

    return poses
```

3. 位移移动（`move_by_displacement`）

输入参数详解

```
def move_by_displacement(
    self,
    arm_tag: ArmTag,              # 手臂标签
    x: float = 0.0,               # X轴位移 (米)
    y: float = 0.0,               # Y轴位移 (米)
    z: float = 0.0,               # Z轴位移 (米)
    quat: list = None,            # 四元数旋转 [qx, qy, qz, qw]
    move_axis: Literal["world", "arm"] = "world",  # 移动坐标系
):
```

内部执行流程

```
def move_by_displacement(self, arm_tag, x=0.0, y=0.0, z=0.0, quat=None, mo
ve_axis="world"):
    # 1. 获取当前手臂姿态
    if arm_tag == "left":
        origin_pose = np.array(self.robot.get_left_ee_pose(), dtype=np.float64)
    elif arm_tag == "right":
        origin_pose = np.array(self.robot.get_right_ee_pose(), dtype=np.float64)

    # 2. 计算位移
    displacement = np.zeros(7, dtype=np.float64)

    if move_axis == "world":
        # 世界坐标系下的位移
        displacement[:3] = np.array([x, y, z], dtype=np.float64)
    else:
        # 手臂坐标系下的位移
        dir_vec = transforms._toPose(origin_pose).to_transformation_matrix()[:3,
0]
        dir_vec /= np.linalg.norm(dir_vec)
        displacement[:3] = -z * dir_vec  # 沿手臂方向移动

    # 3. 计算新姿态
    new_pose = origin_pose + displacement

    # 4. 应用旋转（如果有）
    if quat is not None:
        new_pose[3:] = quat

    # 5. 生成动作
    return arm_tag, [Action(arm_tag, "move", target_pose=new_pose)]
```

4. 动作执行系统（`move`）

输入参数详解

```
def move(
    self,
    actions_by_arm1: tuple[ArmTag, list[Action]],  # 第一个手臂的动作
    actions_by_arm2: tuple[ArmTag, list[Action]] = None,  # 第二个手臂的动作
    save_freq=-1,  # 保存频率
):
```

内部执行流程

```
def move(self, actions_by_arm1, actions_by_arm2=None, save_freq=-1):
    # 1. 检查规划状态
    if self.plan_success is False:
        return False

    # 2. 解析动作
    actions = [actions_by_arm1, actions_by_arm2]
    left_actions = get_actions(actions, "left")
    right_actions = get_actions(actions, "right")

    # 3. 同步动作长度
    max_len = max(len(left_actions), len(right_actions))
    left_actions += [None] * (max_len - len(left_actions))
    right_actions += [None] * (max_len - len(right_actions))

    # 4. 执行动作序列
    for left, right in zip(left_actions, right_actions):
        # 执行左臂动作
        if left is not None:
            self.take_action(left)

        # 执行右臂动作
        if right is not None:
            self.take_action(right)
```

```
        # 保存数据（如果需要）
        if save_freq != None:
            self._take_picture()
```

5. 动作执行（`take_action`）

内部执行流程

```
def take_action(self, action: Action):
    # 1. 解析动作类型
    if action.action == "move":
        # 移动动作
        target_pose = action.target_pose

        # 2. 运动规划
        if action.arm_tag == "left":
            success = self.robot.left_planner.plan_to_pose(target_pose)
            if success:
                # 3. 执行轨迹
                control_seq = self.robot.left_planner.get_control_sequence()
                self.take_dense_action(control_seq)
        else:
            success = self.robot.right_planner.plan_to_pose(target_pose)
            if success:
                control_seq = self.robot.right_planner.get_control_sequence()
                self.take_dense_action(control_seq)

    elif action.action == "gripper":
        # 夹爪动作
        target_pos = action.target_gripper_pos

        if action.arm_tag == "left":
            self.robot.set_left_gripper_position(target_pos)
        else:
            self.robot.set_right_gripper_position(target_pos)
```

6. 密集动作控制（`take_dense_action`）

输入参数详解

```
def take_dense_action(self, control_seq, save_freq=-1):
    """
    control_seq: {
        "left_arm": {
            "position": np.array,    # 关节位置序列 [N, 7]
            "velocity": np.array,    # 关节速度序列 [N, 7]
        },
        "right_arm": {
            "position": np.array,    # 关节位置序列 [N, 7]
            "velocity": np.array,    # 关节速度序列 [N, 7]
        },
        "left_gripper": {
            "num_step": int,       # 步数
            "target_pos": float,    # 目标位置
        },
        "right_gripper": {
            "num_step": int,       # 步数
            "target_pos": float,    # 目标位置
        }
    }
    """
```

内部执行流程

```
def take_dense_action(self, control_seq, save_freq=-1):
    # 1. 提取控制序列
    left_arm = control_seq.get("left_arm")
    right_arm = control_seq.get("right_arm")
    left_gripper = control_seq.get("left_gripper")
    right_gripper = control_seq.get("right_gripper")
```

```
    # 2. 计算最大控制长度
    max_control_len = 0
    if left_arm is not None:
        max_control_len = max(max_control_len, left_arm["position"].shape[0])
    if right_arm is not None:
        max_control_len = max(max_control_len, right_arm["position"].shape[0])
    if left_gripper is not None:
        max_control_len = max(max_control_len, left_gripper["num_step"])
    if right_gripper is not None:
        max_control_len = max(max_control_len, right_gripper["num_step"])

    # 3. 执行控制序列
    for control_idx in range(max_control_len):
        # 设置关节位置
        if left_arm is not None and control_idx < left_arm["position"].shape[0]:
            self.robot.set_left_joint_positions(left_arm["position"][control_idx])

        if right_arm is not None and control_idx < right_arm["position"].shape[0]:
            self.robot.set_right_joint_positions(right_arm["position"][control_idx])

        # 设置夹爪位置
        if left_gripper is not None and control_idx < left_gripper["num_step"]:
            self.robot.set_left_gripper_position(left_gripper["target_pos"])

        if right_gripper is not None and control_idx < right_gripper["num_step"]:
            self.robot.set_right_gripper_position(right_gripper["target_pos"])

        # 物理仿真步进
        self.scene.step()

        # 保存数据（如果需要）
        if save_freq != None and control_idx % save_freq == 0:
            self._take_picture()
```

🔄 执行流程图

```
用户调用 grasp_actor()
   ↓
choose_grasp_pose() 计算抓取姿态
   ↓
生成 Action 序列
   ↓
move() 执行动作
   ↓
take_action() 处理每个动作
   ↓
运动规划器计算轨迹
   ↓
take_dense_action() 执行密集控制
   ↓
机器人执行动作
   ↓
物理仿真步进
   ↓
返回执行结果
```

这个接口设计提供了从高级任务描述到低级关节控制的完整抽象层次，使得用户可以专注于任务逻辑而不需要关心底层的运动规划和执行细节。

🔄 迁移到现实环境的修改方案

1. 硬件接口层

```
# 现实环境需要实现的接口
class RealRobotInterface:
    def get_joint_positions(self, arm_tag):
        """获取关节位置"""
        pass

    def set_joint_positions(self, arm_tag, positions):
```

```
        """设置关节位置"""
        pass

    def get_end_effector_pose(self, arm_tag):
        """获取末端执行器姿态"""
        pass

    def set_gripper_position(self, arm_tag, position):
        """设置夹爪位置"""
        pass
```

2. 安全系统

```
class SafetySystem:
    def check_collision(self, target_pose):
        """碰撞检测"""
        pass

    def emergency_stop(self):
        """紧急停止"""
        pass

    def force_limit_check(self, force_threshold):
        """力限制检查"""
        pass
```

3. 传感器集成

```
class SensorIntegration:
    def get_camera_image(self):
        """获取相机图像"""
        pass

    def get_depth_image(self):
        """获取深度图像"""
```

```
        pass

    def get_force_torque(self, arm_tag):
        """获取力/力矩传感器数据"""
        pass
```

4. 通信协议

```
# ROS2 接口示例
class ROS2Interface:
    def __init__(self):
        self.joint_publisher = self.create_publisher(...)
        self.gripper_publisher = self.create_publisher(...)
        self.joint_subscriber = self.create_subscription(...)
```

5. 关键修改点

1. **仿真器替换**: 将Sapien物理引擎替换为真实硬件控制
2. **规划器适配**: 适配真实机器人的运动学模型
3. **安全机制**: 添加碰撞检测和力控制
4. **通信协议**: 实现与真实机器人的通信接口
5. **传感器融合**: 集成视觉、力觉等传感器
6. **误差补偿**: 处理定位误差和机械误差

这个基础操作包提供了完整的双臂机器人控制框架，可以支持复杂的操作任务，并且具有良好的可扩展性。

持续debug中:

1. 修改test_gen_code中保存视频的代码，目的是为了找到不同seed simulation失败的原因
2. 找到print simulate失败的地方与控制seed数量的代码
3. 。。。